Distributed key-value store in Akka

Luca dell'Oglio

July 3, 2020

MTDS course, Politecnico di Milano

Introduction

The store has two primitives: `put(K,V)` and `get(K)`.

The store should support scaling by partitioning the key-space and by assigning different keys to different nodes.

The store should store each data element into R replicas to tolerate up to $R - 1$ simultaneous failures without losing any information.

Upon the failure of a node, the data it stored is replicated to a new node to ensure that the system has again R copies.

New nodes can be added to the store dynamically.

Architecture

The store is implemented using the Akka clustering service. The system is composed by:

- Multiple `NodeActor`, which store (`key`,`value`) pairs;
- A `SupervisorActor`, which routes the messages to the required `NodeActor`.

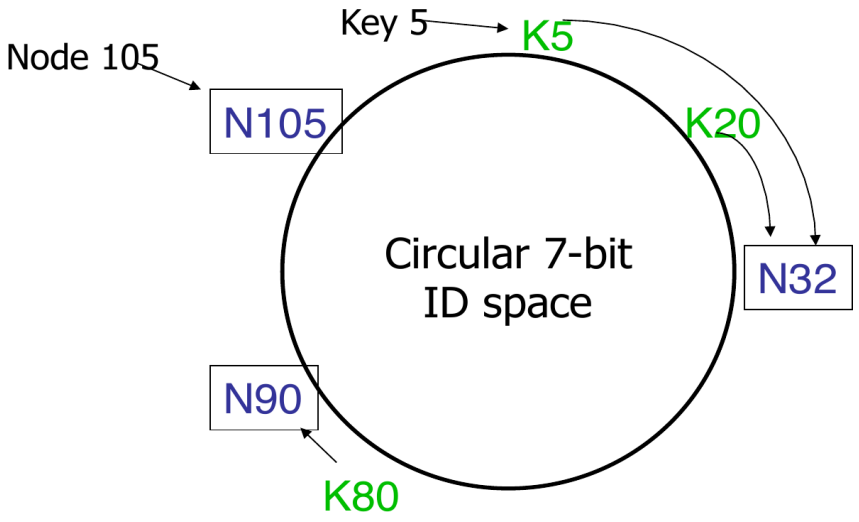
Node IDs

Each node is identified by a `NodePointer`.

```
class NodePointer {
    String address = node.clusterAddress;
    UInt id = hash(node.uniqueAddress);
}
```

The same 32 bit hash function is used to obtain key IDs and node IDs.

Consistent hashing



IDs are ordered on an ID ring modulo 2^m. In this case, $m = 32$.

Key k is assigned to the first node whose ID is equal to or follows k in the identifier space. This node is called the *successor node* of key k.

If identifiers are represented as a circle of numbers from 0 to 2^{m-1}, then `succ(k)` is the first node clockwise from k.

Replication

In addition to `succ(k)`, each entry `(k,v)` is stored on the $R-1$ nodes succeding the key, for a total of R replicas.

A `get` request will try to retrieve the data from `succ(k)` first. If the request fails, it's forwarded to the next $R-1$ nodes.

A `put` request is performed on `succ(k)`, and then the store will handle the replication separately.

Implementation

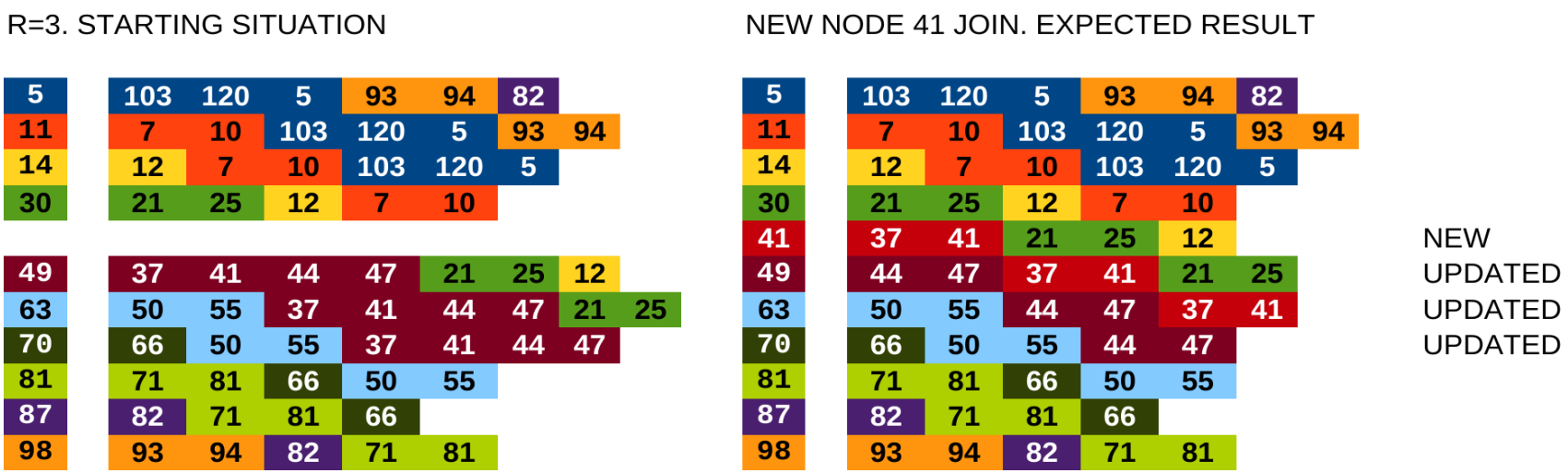
`SupervisorActor` stores the nodes routing information as a `TreeSet` of `NodePointer`. The elements of the `TreeSet` are ordered in ascending order according to the `id` of each `NodePointer`.

- Adding or removing a node and finding `succ(k)` are guaranteed to be $O(\log n)$ operations.
- Set elements are ordered naturally.

Node join

When a node joins:

- `SupervisorActor` adds the new node to the `TreeSet`;
- The new node receives the entries it has to store;
- The R nodes that follow the new node clean the old keys that they don't have to store anymore.



Entries in the new node

`newNode` has to receive its keys from two nodes:

- The keys s.t. `succ(k) = newNode` from its successor (in the example, `K37, K41` from `N49`);
  - `SupervisorActor` sends a `NewPredecessorMsg` to `N49`. When `N49` receives the message, it sends the required keys to `N41` (i.e., the keys for which `N41` is now directly responsible for).
- The replica keys from its predecessor (in the example, `K21, K25, K12` from `K30`).
  - See next slide.

Entries management

To keep the store consistent, `SupervisorActor` periodically sends each `NodeActor` two types of messages:

- An `UpdateSuccessorsMsg`. When a `NodeActor` receives this message, it sends the keys for which it is directly responsible for to the first $R - 1$ nodes that follows it.
- A `CleanKeysMsg`. When a `NodeActor` receives this message, it deletes any key that doesn't belong to its $R - 1$ predecessor or for which it is not directly responsible for.

Node removal

When a node is removed:

- `SupervisorActor` removes the new node to the `TreeSet`;
- The store eventually becomes consistent again, thanks to the periodic messages.